XML or JSON: Guidelines for what to choose for DB2 for z/OS

Jane Man
Susan Malaika

March 27, 2014

IBM® DB2® for z/OS® offers document storage support for both JSON and XML. It is not always apparent whether JSON or XML is most suitable for a particular application. This article provides guidelines to help you select XML or JSON. It includes examples of creating, querying, updating, and managing in both JSON and XML in DB2 for z/OS.

Introduction

IBM DB2 for z/OS offers document storage support for both JavaScript Object Notation (JSON) and Extensible Markup Language (XML). Sometimes it’s hard to know whether JSON or XML is the better choice for a particular application. This article provides guidelines to help you select XML or JSON. Examples walk through creating, querying, updating, and managing with both JSON and XML in DB2 for z/OS.

What is DB2 for z/OS pureXML?

DB2 pureXML supports the storage, validation, and search capability for XML data since DB2 9 for z/OS. You no longer need to store XML data in CLOB or BLOB columns or shred the XML data to fields and store it in relational columns. You can directly insert an XML document into an XML column. The XML data is physically stored in an XML table, not in the base table. (See Related topics for more information about DB2 for z/OS pureXML.)

Listing 1 shows a simple example of an XML document that represents a purchase order (PO) with two items.
Listing 1. Simple example of XML document

```xml
<PO id="123" orderDate="2013-11-18">
  <customer cid="999"/>
  <items>
    <item partNum="872-AA">
      <productName>Lawnmower</productName>
      <quantity>1</quantity>
      <USPrice>149.99</USPrice>
      <shipDate>2013-11-20</shipDate>
    </item>
    <item partNum="945-ZG">
      <productName>Sapphire Bracelet</productName>
      <quantity>2</quantity>
      <USPrice>178.99</USPrice>
      <comment>Not shipped</comment>
    </item>
  </items>
</PO>
```

What is JSON?

JSON is a lightweight data format specified in IETF RFC 4627 that is based on a subset of the JavaScript programming language. JSON is a text format that is easy for humans to read and write and easy for machines to parse and generate. With the increased popularity of JavaScript and the simplicity of JSON, JSON has become popular for presenting information to JavaScript clients.

With preconditioning PTF PM97869, PTF PM98357 enables support for JSON in DB2 10 for z/OS via the Optim client API. The enabling PTF adds several DB2-supplied user defined functions (UDFs), a DB2-supplied stored procedure, and a database for JSON support. All of these objects are created in the SYSTOOLS schema. Refer to DB2 for z/OS info APAR II14727 and ++HOLD of PM98357 (in Related topics) for more information.

Listing 2 is a simple example of a JSON document using the same PO as in Listing 1.

Listing 2. Simple example of a JSON document

```json
{
  "PO": {
    "@id": 123,
    "@orderDate": "2013-11-18",
    "customer": { "@cid": 999 },
    "items": [
      {
        "@partNum": "872-AA",
        "productName": "Lawnmower",
        "quantity": 1,
        "USPrice": 149.99,
        "shipDate": "2013-11-20"
      },
      {
        "@partNum": "945-ZG",
        "productName": "Sapphire Bracelet",
        "quantity": 2,
        "USPrice": 178.99,
        "comment": "Not shipped"
      }
    ]
  }
}
```
Selecting XML or JSON for an application

As shown in the code listings, XML and JSON are both serialized formats for structured data and offer similar capabilities. In this section, we distinguish XML from JSON in general and with respect to DB2 in particular.

Both XML and JSON:

- Make it possible to describe hierarchical data structures that include character strings and numeric values.
- Have mechanisms for repetitions and are suited for documents that incorporate business data.
- Came about in the final years of the 20th century when World Wide Web applications were gaining significant use in businesses.
- Are well-suited for use in volatile environments where data structures change often.

In XML, a typical example of data is an insurance policy or an invoice. In JSON, a typical example is state information for an application.

XML is sophisticated and has many mechanisms that might not always be needed for every application. The following are capabilities specific to XML.

**Namespaces**

Different groups can define and regulate different sections of a document with namespaces. For example, an institution that defines data exchange formats for banks can provide its definitions within the context of a particular public namespace. A bank can define its own internal extensions within the context of its own internal namespace. The separate sections of the document are clearly delineated and can be managed and regulated separately.

**Schemas**

Documents can be validated against a particular structure, and the content checked against constraints, with schemas. The schema language is declarative and associated with a namespace. Thus, owners of a namespace can make schemas available to govern or regulate the expected structure. Schemas can evolve over time as the needs of the business change. As they evolve, new namespaces are associated with new schema versions.

**Transformation Language (XSL)**

You can transform one XML shape into another textual document with XSL. The language is declarative.

**Query Language (XQuery)**

You can query a collection of XML documents with XQuery. The language is declarative. An element of the XQuery language is XPath, which defines portions of an XML document.
arrays can be defined in a data structure and accessed via array subscript. There are two main data types—numeric and string—although some systems have introduced their own data type extensions, such as date. Standardized schema, query, and transformation languages have not yet emerged for JSON, but individual products have introduced such languages.

Table 1 summarizes some of the standardized interfaces for relational, XML, and JSON data.

<table>
<thead>
<tr>
<th>Property</th>
<th>Relational</th>
<th>JSON</th>
<th>XML</th>
</tr>
</thead>
<tbody>
<tr>
<td>Metadata</td>
<td>Data Definition Language (ISO)</td>
<td></td>
<td>XML Schema XSD (W3C), Namespaces (W3C)</td>
</tr>
<tr>
<td>Query &amp; CRUD languages</td>
<td>Data Manipulation Language (DML), SQL, SQL/XML (ISO)</td>
<td></td>
<td>XPath, XQuery (W3C)</td>
</tr>
<tr>
<td>Transformation &amp; other languages</td>
<td>SQL (Tables to Tables)</td>
<td>JavaScript</td>
<td>XSLT (XML to text, includes XML), XForms SQL XMLTABLE (XML to relational)</td>
</tr>
</tbody>
</table>

Both XML and JSON make schema evolution simple and coexist with relational data. As shown in Figure 1, typical use cases for XML and JSON are as follows.

- XML is suitable for data exchange or sharing between independent entities, systems, or applications, particularly where the domain is regulated. It allows third parties to define portions of data structures independently (for example, banking or insurance).
- JSON is suitable for use for data exchange or sharing within an application. It's typically used with human interfaces and mobile applications, making it straightforward to pass data structures back and forth.

Figure 1. XML and JSON common use cases

You can store, index, query, and update XML and JSON in DB2. There is strong integration between XML and relational support in DB2 through SQL/XML. JSON is accessed through the
DB2 JSON API and through user defined functions. Both XML and JSON in DB2 are suitable for environments where data structures and schemas change frequently, in contrast with relational data where data structure changes tend to happen less frequently. As long as the XML is well-formed, it can be stored in the same collection (column) even if the structure of its content is different from what's stored there already. The same is true of JSON.

The following sections provide examples of creating, querying, updating, and managing JSON and XML in DB2 for z/OS. The focus is on areas that are common to both XML and JSON. If you're looking for a full set of features and functions of individual data formats, see Related topics.

Creating XML and JSON tables with indexes

This section discusses how to create XML and JSON tables with indexes.

**XML column**

Starting with DB2 9 for z/OS, you can directly create a table with a column of XML datatype. Listing 3 shows how to create a table called XMLT1 with an XML column called XMLPO.

**Listing 3. Create a table with XML column**

```sql
CREATE STOGROUP STGP1 VOLUMES(SCR03) VCAT DSNCAT;
CREATE DATABASE DB1;
CREATE TABLESPACE TS1 IN DB1 USING STOGROUP STGP1...
CREATE TABLE XMLT1 (ID INT, XMLPO XML) IN DB1.TS1;
```

In DB2 10 for z/OS, you can associate one or more XML schemas with an XML column such that schema validation is implicitly executed during insert, update, and LOAD to the XML column.

Suppose SYSXSR.PO1 is a registered XML schema in DB2 for z/OS. Listing 4 has an example of creating a table with XML column associated with an XML schema.

**Listing 4. Create a table with XML column associated with an XML schema**

```sql
CREATE TABLE XMLT2 (ID INT, XMLPO XML(XMLSCHEMA ID SYSXSR.PO1)) IN DB1.TS1;
```

**XML Index**

To improve performance during SELECT, you can create an XML index on customer id (for the XML document in Listing 1), as in Listing 5.

**Listing 5. Create an XML index**

```sql
create index custidx1 on XMLT1(XMLPO)
generate key using
xmlpattern '/PO/customer/@cid'
as sql decfloat
```

If you use display database for DB1, in addition to **TS1** and **CUSTIDX1** that you explicitly create, DB2 also implicitly creates a tablespace called **XXML0000** (to store the XML records) and two
indexes (IRDOCIDX and IRNODEID). Listing 6 shows an example. The name of implicitly created objects might not be the same each time.

**Listing 6. Display database**

```
@DISPLAY DATABASE(DB1)
DSNT360I @ ********************************************
DSNT361I @ * DISPLAY DATABASE SUMMARY 110
*    GLOBAL
DSNT360I @ ********************************************
DSNT362I @ DATABASE = DB1 STATUS = RW 112
DBD LENGTH = 4028
DSNT397I @ 113
NAME TYPE PART STATUS PHYERRLO PHYERRHI CATALOG PIECE
-------- ---- ---- ------ -------- -------- ------- ----- 
TS1 TS 0001 RW
XXML0000 XS 0001 RW
CUSTIDX1 IX L0001 RW
IRDOCIDX IX L0001 RW
IRNODEID IX L0001 RW
******* DISPLAY OF DATABASE DB1 ENDED ***************
```

**JSON**

You cannot use SQL to create a table to store JSON data directly. Unlike XML, there is no JSON data type. The JSON data are stored internally in BSON (binary format of JSON) in the inline BLOB column.

The infrastructure for JSON is created by the installation job DSNTIJRT, which has been modified in PM98357. You need to customize this installation job and execute it to create the following objects for JSON support.

- UDFs
- 1 stored procedure: REGSP
- 1 table: SYSJSON_INDEX. It has an index called SYSJSON_INDEX_IX1 and is created in a database called DSN5JSDB and a table space called DSN5JSTS.

All of these objects are created in the SYSTOOLS schema.

**JSON namespace**

In DB2 10 for z/OS, you can store the JSON document under a specific JSON namespace, which is similar to a DB2 SQL schema name. Listing 7 shows how to set TEST as a JSON namespace using the command-line processor (CLP). In Listing 7, you can use `db` to see the current database connection.

**Listing 7. Set TEST as a JSON namespace**

```
nosql>db
Database: jdbc:db2://dtec221.vmec.svl.ibm.com:446/STLEC1 Schema: TEST
nosql>use TEST
Switched to schema: TEST
```
For all of the JSON commands in this article we're using the DB2 JSON command-line interface. You can do similar tasks using the NoSQL JSON API. See Related topics for more information.

**JSON collection**

JSON documents are organized in collections, which under the covers are similar to relational tables. Listing 8 shows how to explicitly create a JSON collection called JSONPO.

**Listing 8. Create a JSON collection explicitly**

```javascript
nosql>db.createCollection("JSONPO", {_id: "$oid"})
Collection: TEST."JSONPO" created. Use db.JSONPO.
```

If JSON trace is on, you will see the corresponding SQL statement that is sent to DB2.

```
[nosql][2013-12-24-20:36:52.653][main][Debug][out] CREATE TABLE
TEST."JSONPO" (ID CHAR(12) FOR BIT DATA NOT NULL, DATA BLOB(16M) INLINE LENGTH 25000, PRIMARY KEY(ID)) CCSID UNICODE
```

From this trace, a table called TEST."JSONPO" is created implicitly with an ID column of CHAR(12) FOR BIT DATA and an INLINE BLOB(16M) column of inline length 25000.

However, it is not necessary to define a table structure for collections. To insert a JSON document, you only need to specify the collection name. If the collection does not exist, it will be automatically created.

**JSON index**

To create an index called myJSONIndex on the `customer cid` field in ascending order, you can use `ensureIndex`, as in Listing 9.

**Listing 9. Create a JSON index**

```javascript
nosql>db.createCollection("JSONPO", {_id: "$oid"})
nosql>db.JSONPO.ensureIndex({"PO.customer.@cid":[1, "$int"]}, "myJSONIndex")
Index <myJSONIndex> was created successfully.
```

If JSON trace is on, you will see the following trace.

```
[nosql][2013-12-25-10:14:38.801][main][Debug][out] CREATE INDEX
TEST."myJSONIndex" ON TEST."JSONPO" ( JSON_VAL(DATA, 'PO.customer.@cid', 'f:na'))
```

The JSON index is created using "index on expression." `JSON_VAL()` is used to extract SQL values from BLOB(BSON).

Similarly, you can create a JSON index with different options such as in descending order, with different data type and field length (for string), and so on.

Listing 10 shows how to retrieve all the index information on the JSONPO collection.
Listing 10. Retrieve JSON index on a collection

```plaintext
nosql>db.JSONPO.getIndexes()
[{
"v":0,"_id":6,"key":{"_id":1},"ns":"TEST.JSONPO","name":"_id_","unique":true,
"datatype":"CHAR(12) FOR BIT DATA","datalength":25000,"idxtype":"typemarker"},
{
"v":0,"_id":9,"key":{"PO.customer.@cid":1},"ns":"TEST.JSONPO","name":"my JSONInde x","unique":false,"datatype":"INTEGER","datalength":1024,"idxtype":"func tion_na"}
]
```

In addition to Index myJSONIndex that was created explicitly in Listing 10, a unique index is automatically created on the _id attribute. A SELECT from the catalog table (SYSIBM.SYSINDEXES) shows a similar result in Listing 11.

Listing 11. Retrieve JSON index info from Catalog table

```sql
select name, tbname from SYSIBM.SYSINDEXES where tbname='JSONPO'
NAME TBNAME
JSONPOAB_#_FSD JSONPO
myJSONIndex JSONPO
2 record(s) selected
```

Storing data

This section discusses storing data with XML and JSON.

**XML**

You can insert XML data directly into a table, as in Listing 12, or you can insert from a file. Just as with the other datatype, you can insert XML data using the LOAD utilities (from sysRec or from the file reference variable).

The parsing of XML data is done inside DB2. DB2 will throw a negative SQLCODE when there is a parsing error. After successful parsing, DB2 will group the XML data in 16k records and store them in an XML tablespace (not in the base table).

Listing 12. Insert XML data

```xml
INSERT INTO XMLT1 values(1,
'<PO id="123" orderDate="2013-11-18">
  <customer cid="999"/>
  <items>
    <item partNum="872-AA">
      <productName>Lawnmower</productName>
      <quantity>1</quantity>
      <USPrice>149.99</USPrice>
      <shipDate>2013-11-20</shipDate>
    </item>
    <item partNum="945-ZG">
      <productName>Sapphire Bracelet</productName>
      <quantity>2</quantity>
      <USPrice>178.99</USPrice>
      <comment>Not shipped</comment>
    </item>
  </items>
</PO>')</n```
JSON

You can insert JSON data directly into DB2 as shown in Listing 13. The syntax of the command is:

```java
db.<collection_name>.insert(<document>)
```

Similarly, you can import JSON documents from a file. The JSON data is converted to BSON before sending to DB2.

**Listing 13. Insert JSON data (formatted for readability)**

```
nosql>db.JSONPO.insert(
    {
        "PO": {
            "@id": 123,
            "@orderDate": "2013-11-18",
            "customer": { "@cid": 999 },
            "items": {
                "item": [
                    {
                        "@partNum": "872-AA",
                        "productName": "Lawnmower",
                        "quantity": 1,
                        "USPrice": 149.99,
                        "shipDate": "2013-11-20"
                    },
                    {
                        "@partNum": "945-ZG",
                        "productName": "Sapphire Bracelet",
                        "quantity": 2,
                        "USPrice": 178.99,
                        "comment": "Not shipped"
                    }
                ]
            }
        }
    })
```

In the `nosql` trace in Listing 14 you can see the `INSERT` statement. There is a logging of `WriteConcern`, which illustrates the setting for the “Fire and Forget” feature. The Fire and Forget feature is designed for performance purposes, but the downside to using this mode is that the data is not guaranteed to be written to the database.

**Listing 14. nosql trace**

```
[nosql][2013-12-25-09:20:34.125][main][DBCollection@37bc37bc][insert] ENTRY {"PO":{"@id":123,"@orderDate":"2013-11-18","customer":{"@cid":999},"items":{"item":[{"@partNum":"872-AA","productName":"Lawnmower","quantity":1,"USPrice":"149.99","shipDate":"2013-11-20"},
                    {
                        "@partNum": "945-ZG",
                        "productName": "Sapphire Bracelet",
                        "quantity": 2,
                        "USPrice": 178.99,
                        "comment": "Not shipped"
                    }
                ]
            }}
WriteConcern {"getlasterror":1,"w":1,"wtimeout":0,"j":true} / (Continue Inserting on Errors? false)
[nosql][2013-12-25-09:20:34.177][main][Debug][out] INSERT INTO TEST."JSONPO" (ID,DATA) VALUES(?, ?)
```

In the JCC trace in Listing 15 you can see the JSON data is converted to BSON before sending to DB2.
If you pay attention to the MVS console, you'll notice that the Work Load Manager (WLM) you declared for the JSON UDFs (in DSNTIJRT installation job) is started while you insert the JSON document.

09.19.07 STC00872 $HASP100 VA1AWLMG ON STCINRDR - 09.19.07 STC00872 $HASP373 VA1AWLMG
STARTED

### Querying data

This section explores querying data for XML and JSON.

#### XML

In DB2 for z/OS you use SQL/XML to query XML data. For more complicated queries, you can use XQuery inside SQL/XML to query and construct data, as shown in the examples below.

You can retrieve the whole XML document for the PO with `customer cid` equal to 999, as in Listing 16. `XMLExists()` is used to filter the XML document we want.

**Listing 16. Retrieve whole XML document**

```
SELECT XMLPO
FROM XMLT1
WHERE XML EXISTS('/PO/customer[@cid=999]' PASSING XMLPO)
```

If you run EXPLAIN, as in Listing 17, the XML index CUSTIDX1 that we created above is used to retrieve the XML data.
Listing 17. EXPLAIN statement and its output for Retrieve whole XML document

```sql
EXPLAIN ALL SET QUERYNO = 1 FOR
SELECT XMLPO
FROM XMLT1
WHERE XML_EXISTS('/PO/customer[@cid=999]' PASSING XMLPO);
```

```
SELECT QUERYNO, QBLOCKNO, PLANNO, METHOD, SUBSTR(TNAME, 1, 8) AS TNAME, TABNO, ACCESSTYPE, SUBSTR(ACCESSNAME, 1, 15) AS ACCESSNAME
FROM PLAN_TABLE
WHERE QUERYNO = 1 ORDER BY 1, 2, 3
```

<table>
<thead>
<tr>
<th>QUERYNO</th>
<th>QBLOCKNO</th>
<th>PLANNO</th>
<th>METHOD</th>
<th>TNAME</th>
<th>TABNO</th>
<th>ACCESSTYPE</th>
<th>ACCESSNAME</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>XMLT1</td>
<td>1</td>
<td>DX</td>
<td>CUSTIDX1</td>
</tr>
</tbody>
</table>

1 record(s) selected

You can use `XMLQuery()` to retrieve part of XML data. The example in Listing 18 uses `XMLQuery` to get the `productName` for customer `cid` equal to 999.

### Listing 18. Use XMLQuery to retrieve part of XML document

```sql
SELECT XMLQuery('/PO/items/item/productName' PASSING XMLPO)
FROM XMLT1
WHERE XML_EXISTS('/PO/customer[@cid=999]' PASSING XMLPO)
```

Result:

```xml
<productName>Lawnmower</productName><productName>Sapphire Bracelet</productName>
```

1 record(s) selected

You can also use `XMLQuery` to retrieve and construct a new XML document. Listing 19 shows how to find the total cost for the customer `cid` equal to 999.

### Listing 19. Use XQuery to find total cost of a PO

```xml
SELECT XMLQuery( 'let $cost:=('
  for $i in /PO/items/item
  let $p:=$i/USPrice
  let $q:=$i/quantity
  where xs:decimal($p)>0 and xs:integer($q)>0 return $p*$q
  ) return fn:sum($cost)
',
PASSING XMLPO)
FROM XMLT1
WHERE XML_EXISTS('/PO/customer[@cid=999]' PASSING XMLPO)
```

The result is 507.97.

Listing 19 shows the use of a FOR loop to loop through each item in the PO. For each item, for those `USPrice` and `quantity` that are larger than 0, multiply them to get individual item cost. Finally, apply `fn:sum()` to sum up all the individual item costs.

In general, XQuery is very powerful for querying XML data. In addition to FLWOR constructs (For, Let, Where, Order By, Return) that are similar to Listing 19, you can also use conditional
expression (IF..THEN..ELSE), value comparison (eq, ne, lt, le, gt, ge), node comparison (is, <<, >>), castable expression, and XQuery constructors that query XML data and construct XML data on the fly.

**JSON**

There are options to count, find, and aggregate data in collections. You can select specific attributes by specifying the conditions that need to be matched. There are control options that let you sort the results (sort()), limit the number of results (limit()), page through a larger set of results (skip()), find the first matching document (findOne()), get the total count of documents in the collection (count()), find distinct values (distinct()), and so on.

You can use find() to query a JSON document with the following syntax.

```

db.<collection>.find(<conditions>, <projection list>)
```

Listing 20 shows how to use the find command to retrieve the whole JSON document for the PO with customer cid equal 999.

**Listing 20. Use find() to find a particular JSON document**

```
nosql>db.JSONPO.find({"PO.customer.@cid": 999})
nosql>Row 1:
nosql> {
    "_id":{"$oid":"52bb80f8564080c0b628c411"},
    "PO":
    {
        "@id":123,
        "@orderDate":"2013-11-18",
        "customer":{"@cid":999},
        "items":
        {
            "item":[
                {
                    "@partNum":"872-AA",
                    "productName":"Lawnmower",
                    "quantity":1,
                    "USPrice":149.99,
                    "shipDate":"2013-11-20"
                },
                {
                    "@partNum":"945-ZG",
                    "productName":"Sapphire Bracelet",
                    "quantity":2,
                    "USPrice":178.99,
                    "comment":"Not shipped"
                }
            ]
        }
    }
}
nosql>1 row(s) returned in 516 milliseconds.
```

If the nosql trace is on, you can see the SELECT statement that is sent to DB2. Note that the JSON_VAL() function is used here.
If you do an EXPLAIN on this SELECT statement, you'll find that the JSON index we created above, myJSONIndex, is used.

EXPLAIN ALL SET QUERYNO = 3 FOR
SELECT ID, DATA FROM TEST."JSONPO" WHERE (JSON_VAL(DATA, 'PO.customer.@cid', 'f:na')=?)

EXPLAIN QUERYNO, QBLOCKNO, PLANNO, METHOD, SUBSTR(TNAME,1,8) AS TNAME,
TABNO, ACCESSTYPE, SUBSTR(ACCESSNAME,1,15) AS ACCESSNAME
FROM PLAN_TABLE
WHERE QUERYNO = 3
ORDER BY 1, 2, 3

QUERYNO    QBLOCKNO  PLANNO  METHOD  TNAME    TABNO  ACCESSTYPE  ACCESSNAME
3          1         1       0       JSONPO   1      I          myJSONIndex

1 record(s) selected

If you're only interested in retrieving certain attributes, say productName for customer cid equal to 999, you can use find() with a projection list, as in Listing 21. We only include the productName and exclude the _id attribute.

Listing 21. Use find() to find certain attributes in a JSON document

nosql>db.JSONPO.find({"PO.customer.@cid": 999}, {"_id":0, "PO.items.item.productName":1})
nosql>Row 1:
{ "items": [ { "productName": "Lawnmower" }, { "productName": "Sapphire Bracelet" } ] }
nosql>1 row(s) returned in 214 milliseconds.

If the nosql trace is on, you can see the SELECT statement that is sent to DB2:

[noSQL][2014-01-07-10:36:35.066][main][Debug][out] SELECT CAST(SYSTOOLS.JSON_BINARY2(DATA, 'PO.items.item.productName', 2048) AS VARCHAR(2048) FOR BIT DATA) AS "xPO_items_item_productName" FROM TEST."JSONPO" WHERE (JSON_VAL(DATA, 'PO.customer.@cid', 'f:na')=?)

From the JCC trace in Listing 22 you can tell that only the required fields, not the entire JSON document, are sent back to the client.

Listing 22. JCC trace to see the required fields sent back to client

[0000 0087005300001800 241B00000079FF00 ...S....$....y....]
[0010 0000740373000000 03504F006A000000 ..t.s....PO.j...]
[0020 036974656D73005E 000000046974656D .items.^....item]
[0030 0053000000033000 200000000270726F .S....0. ....pro]
[0040 647563744E616D650012000000 productName....
[0050 53617070686972652042726163656C65 Sapphire Bracele]
For more complicated queries, you can use the `aggregate()` function to describe a query in multiple steps with output of one step providing the pipeline as input to the next step. However, there are some restrictions on data in arrays. Please refer to the latest documentation for updates on these restrictions.

**Updating data**

This section compares how to update data in XML and JSON.

**XML**

Starting with DB10 for z/OS, you can update a whole XML document, as in Listing 23, or just update part of a document (sub-document update) using `XMLModify()`. You can do insert, replace, and delete with `XMLModify()`. Unlike the XML update in other database systems that retrieve the whole XML data from disk, and update and write the whole modified XML document back to disk, `XMLModify()` is doing a “real” sub-document. For example, `XMLModify()` only modifies the records that need to be changed and does not write the whole modified document. This greatly improves the performance when changes are small when compared to the size of the whole document.

**Listing 23. Whole document update**

```sql
-- Update the XML document for id=1
UPDATE XMLT1 SET XMLPO =
'<PO id="123" orderDate="2013-11-18">
  <customer cid="111"/>
  <items>
    <item partNum="872-AA">
      <productName>Lawnmower</productName>
      <quantity>1</quantity>
      <USPrice>149.99</USPrice>
      <shipDate>2013-11-20</shipDate>
    </item>
    <item partNum="945-ZG">
      <productName>Sapphire Bracelet</productName>
      <quantity>2</quantity>
      <USPrice>178.99</USPrice>
      <comment>Not shipped</comment>
    </item>
  </items>
</PO>'
WHERE id=1
```

Let’s look at some `XMLModify()` examples. Listing 24 shows how to add a new item to an existing XML document.

**Listing 24. Use `XMLModify()` to add new nodes**

```sql
-- add 1 more item to the XMLPO for id=1
UPDATE XMLT1 SET XMLPO =
XMLModify('insert node $newItem/item as last into /PO/items',
XMLPARSE(document
'  <item partNum="200-AA">
    <productName>SKII daily lotion</productName>
    <quantity>1</quantity>
    <USPrice>180</USPrice>
    <comment>Not shipped</comment>
  </item>
') as "newItem")
WHERE id=1;
```
The output from Listing 24 (reformatted for readability) is shown below. To verify the result of XMLModify(), execute the following SELECT statement to select the modified XML document:

```sql
select XMLPO from XMLT1 where id=1
```

```
<PO id="123" orderDate="2013-11-18">
  <customer cid="111"/>
  <items>
    <item partNum="872-AA">
      <productName>Lawnmower</productName>
      <quantity>1</quantity>
      <USPrice>149.99</USPrice>
      <shipDate>2013-11-20</shipDate>
    </item>
    <item partNum="945-ZG">
      <productName>Sapphire Bracelet</productName>
      <quantity>2</quantity>
      <USPrice>178.99</USPrice>
      <comment>Not shipped</comment>
    </item>
    <item partNum="200-AA">
      <productName>SKII daily lotion</productName>
      <quantity>1</quantity>
      <USPrice>100</USPrice>
      <comment>Not shipped</comment>
    </item>
  </items>
</PO>
```

Suppose you want to increase the USPrice of the new item we added (SKII daily lotion). You can use the statement in Listing 25.

**Listing 25. Use XMLModify to replace value of a node**

```sql
-- replace the USPrice of SKII daily lotion
UPDATE XMLT1 SET XMLPO = XMLModify('replace value of node /PO/items/item[@partNum="200-AA"]/USPrice with xs:decimal(200)')
Where id=1;
```

The output from the SELECT statement (reformatted for readability) is shown below.

```
<PO id="123" orderDate="2013-11-18">
  <customer cid="111"/>
  <items>
    <item partNum="872-AA">
      <productName>Lawnmower</productName>
      <quantity>1</quantity>
      <USPrice>149.99</USPrice>
      <shipDate>2013-11-20</shipDate>
    </item>
    <item partNum="945-ZG">
      <productName>Sapphire Bracelet</productName>
      <quantity>2</quantity>
      <USPrice>178.99</USPrice>
      <comment>Not shipped</comment>
    </item>
    <item partNum="200-AA">
      <productName>SKII daily lotion</productName>
      <quantity>1</quantity>
      <USPrice>200</USPrice>
      <comment>Not shipped</comment>
    </item>
  </items>
</PO>
```
Now suppose you want to delete the new item (SKII daily lotion) since the price is increased. You can use the `update` statement in Listing 26 to delete nodes.

**Listing 26. Use XMLModify to delete node**

```sql
-- delete SKII daily lotion
UPDATE XMLT1 SET XMLPO =
    XMLModify('delete node /PO/items/item[@partNum="200-AA"]')
Where id=1;
```

The output, reformatted for readability, is shown below.

```sql
select XMLPO from XMLT1 where id=1; XMLPO
<PO id="123" orderDate="2013-11-18">
    <customer cid="111"/>
    <items>
        <item partNum="872-AA">
            <productName>Lawnmower</productName>
            <quantity>1</quantity>
            <USPrice>149.99</USPrice>
            <shipDate>2013-11-20</shipDate>
        </item>
        <item partNum="945-ZG">
            <productName>Sapphire Bracelet</productName>
            <quantity>2</quantity>
            <USPrice>178.99</USPrice>
            <comment>Not shipped</comment>
        </item>
    </items>
</PO>
```

**JSON**

The `update()` function in the DB2 JSON API lets you update or append specific fields or replace the whole document. The CLP syntax is:

```sql
update(<condition>, <fields to update>, <upsert>, <multi>)
```

The `$set` operator should be in the second parameter to update particular fields, but leave the other fields unchanged. Similarly, you can use the `$unset` operator to delete a particular field.

When `<upsert>` is true, the document will be updated if the document exists. Otherwise, it is inserted if the document does not exist.

When `<multi>` is true, all documents matching the query will be updated. If it is set to false, only the first document matching the query will be updated.

Listing 27 shows how we replace a whole JSON document for the document with `PO.customer.@cid=999`.

**Listing 27. Replace a whole JSON document (formatted for readability)**

```sql
nosql>db.JSONPO.update("PO.customer.@cid": 999), {"PO":
    {"id": 123,
```

XML or JSON: Guidelines for what to choose for DB2 for z/OS
"@orderDate": "2013-11-18",
"customer": { "@cid": 111 },
"items": [ 
  "item": [ 
    { "@partNum": "872-AA",
      "productName": "Lawnmower",
      "quantity": 1,
      "USPrice": 149.99,
      "shipDate": "2013-11-20"},
    { "@partNum": "945-ZG",
      "productName": "Sapphire Bracelet",
      "quantity": 2,
      "USPrice": 178.99,
      "comment": "Not shipped" ]
  ]
}
}
}
nosql>db.JSONPO.find()
nosql>Row 1: 
nosql> { 
  "_id":{"$oid":"52bb80f8564080c0b628c411"},
  "PO": 
  { 
    "@id":123,
    "@orderDate":"2013-11-18",
    "customer":{"@cid":111},
    "items": 
    { 
      "item": [ 
        { "@partNum":"872-AA",
          "productName":"Lawnmower",
          "quantity":1,
          "USPrice":149.99,
          "shipDate":"2013-11-20" },
        { "@partNum":"945-ZG",
          "productName":"Sapphire Bracelet",
          "quantity":2,
          "USPrice":178.99,
          "comment":"Not shipped" }
      ]
    }
  }
}
nosql>1 row(s) returned in 479 milliseconds.

Just as we did to the XML document, let's add one more item (SKII daily Lotion) to the JSON document in Listing 28. Using the $set operation ensures that the other fields are unchanged.

Listing 28. Add one rested element to a whole JSON document (formatted for readability)

```sql
nosql>db.JSONPO.update( 
{ "PO.customer.@cid": 111 }, 
{ $set:{"PO.items.item.2": {"@partNum": "200-AA",
  "productName": "SKII daily lotion",
  "quantity": 1,
  "USPrice": 100,
  "comment": "Not shipped"}}
}
)
```
Updated 1 row(s).
	nosql>db.JSONPO.find()
nosql>Row 1:
nosql> {
  
    "_id":{"$oid":"52be06c65640c36104329f3c"},
    "PO":
    {
      "@id":123,
      "@orderDate":"2013-11-18",
      "customer":{"@cid":111},
      "items":
      {
        "item":[
          {
            "@partNum":"872-AA",
            "productName":"Lawnmower",
            "quantity":1,
            "USPrice":149.99,
            "shipDate":"2013-11-20"
          },
          {
            "@partNum":"945-ZG",
            "productName":"Sapphire Bracelet",
            "quantity":2,
            "USPrice":178.99,
            "comment":"Not shipped"
          },
          {
            "@partNum":"200-AA",
            "productName":"SKII daily lotion",
            "quantity":1,
            "USPrice":100,
            "comment":"Not shipped"
          }
        ]
      }
    }
  }
}

If trace is on, you can see that the update statement above is translated to the following SQL
statement with three UDF calls.

[nosql][2013-12-27-14:38:18.643][main][Debug][out] UPDATE TEST."JSONPO"
SET DATA=
SYSTOOLS.JSON_UPDATE(DATA,'$set','items.item.2',?,SYSTOOLS.JSON_GET_POS_ARR_INDEX(DATA,?)) WHERE TEST."JSONPO".ID = (SELECT ID FROM
TEST."JSONPO" WHERE (JSON_VAL(DATA, 'PO.customer.@cid', 'f:na')=?)
FETCH FIRST ROW ONLY)

To increase the USPrice of the new item (SKII daily lotion) from 100 to 200, as in Listing 29, first
set the conditions ("PO.customer.@cid": 111 and "PO.items.item.productName":"SKII daily
lotion"). Then change the USPrice to 200. The $set operation is used here to ensure the other
fields are not changed.
**Listing 29. Replace value of a field in a JSON document (formatted for readability)**

```javascript
nosql>db.JSONPO.update(
    {"PO.customer.@cid": 111,
    "PO.items.item.productName":"SKII daily lotion"},
    { $set:{"PO.items.item.$.USPrice": 200}})
Updated 1 row(s).

nosql>db.JSONPO.find()
nosql>Row 1:

   { "_id":{"$oid":"52be2fae5640c36104329f3d"},
    "PO":
    { "@id":123,
      "@orderDate":"2013-11-18",
      "customer":{"@cid":111},
      "items":
      { "item"[:
          { "@partNum":"872-AA",
            "productName":"Lawnmower",
            "quantity":1,
            "USPrice":149.99,
            "shipDate":"2013-11-20" },
          { "@partNum":"945-ZG",
            "productName":"Sapphire Bracelet",
            "quantity":2,
            "USPrice":178.99,
            "comment":"Not shipped" },
          { "@partNum":"200-AA",
            "productName":"SKII daily lotion",
            "quantity":1,
            "USPrice":200,
            "comment":"Not shipped" } ]
    } }
```

Now, since the price is increased, we want to delete the newly added item (SKII daily lotion). Since we want to delete an array entry, you need to use `$unset` first to change the array entry to null and then use `$pull` to remove the null array entry, as in Listing 30.

**Listing 30. Delete a field in a JSON document (formatted for readability)**

```javascript
nosql>db.JSONPO.update(
    {"PO.customer.@cid": 111,
    "PO.items.item.productName":"SKII daily lotion" },
    { $unset:{"PO.items.item.$":null}})
Updated 1 row(s).

nosql>db.JSONPO.find()
nosql>Row 1:

   { "_id":{"$oid":"52be34195640c36104329f3f"},
```
Utilities support

This section outlines utility support for XML and JSON.
XML

Because XML is a regular data type, almost all utilities are enhanced to support XML. You can also run CHECK DATA to do a structure check and XML schema check on XML data that are stored in the XML column.

Listings 31, 32, and 33 show UNLOAD, LOAD, and CHECK DATA on XML data.

**Listing 31. UNLOAD XML data**

```
UNLOAD TABLESPACE DB1.TS1
PUNCHDDN SYSPUNCH UNLDDN SYSREC SPANNED YES FROM TABLE ADMF001.XMLT1
  (ID INT,
   XMLPO XML ) NOPAD
```

**Listing 32. LOAD XML**

```
LOAD DATA INDDN SYSREC LOG NO RESUME YES
EBCDIC CCSID(00037,00000,00000)
  FORMAT SPANNED YES
  INTO TABLE
  "ADMF001".
  s
  "T1"
  WHEN(00001:00002) = X'0003'  CHAR(1)
  NUMRECS 1
    "DSN_NULL_IND_00001" POSITION( 00003) , "ID"
    POSITION( 00004:00007) INTEGER
    NULLIF(DSN_NULL_IND_00001)=X'FF'
    , "DSN_NULL_IND_00002" POSITION( *) CHAR(1)
    , "XMLPO" POSITION( *) XML PRESERVE WHITESPACE
    NULLIF(DSN_NULL_IND_00002)=X'FF'
```

**Listing 33. CHECK DATA on XML TABLESPACES**

```
CHECK DATA
  TABLESPACE DB1.TS1 SCOPE ALL
  INCLUDE XML TABLESPACES(ALL)
  SHRLEVEL REFERENCE
```

JSON

The JSON index is based on an expression that invokes the built-in JSON_VAL function. At the time of this writing, the following restrictions apply.

- LOAD utility on a table with one or more JSON index defined.
- CHECK DATA utility on a table with one or more JSON index defined on it.
- REORG TABLESPACE SHRLEVEL CHANGE on a tablespace with one or more JSON index defined in it.
- REBUILD INDEX SHRLEVEL CHANGE on a JSON index.
- CHECK INDEX SHRLEVEL CHANGE on a JSON index.

You can use the SQL statement in Listing 34 to find the database and tablespace name for the table (JSONPO) that we created to store the JSON data.
Listing 34. Find database and tablespace name of the JSON table

```sql
SELECT DBNAME, TSNAME FROM SYSIBM.SYSTABLES
WHERE NAME='JSONPO';
```

<table>
<thead>
<tr>
<th>DBNAME</th>
<th>TSNAME</th>
</tr>
</thead>
<tbody>
<tr>
<td>DSN00003</td>
<td>JSONPO</td>
</tr>
</tbody>
</table>

After you have the database and tablespace name you can UNLOAD, as in Listing 35.

Listing 35. UNLOAD JSON data

```sql
UNLOAD TABLESPACE DSN00003.JSONPO
   PUNCHDDN SYSPUNCH UNLDDN SYSREC SPANNED YES FROM TABLE TEST."JSONPO"
   (ID CHAR(12),
    DATA BLOB) NOPAD
```

However, when using the LOAD statement generated from UNLOAD, LOAD still fails because there is a JSON index defined on the table. Listing 36 shows the LOAD statement and its output.

Listing 36. LOAD of the JSON data

```sql
LOAD DATA INDDN SYSREC LOG NO RESUME YES UNICODE CCSID(00367,01208,01200)
   FORMAT SPANNED YES
   INTO TABLE
   "TEST".
   "JSONPO"
   WHEN(00001:00002) = X'0003'
   NUMRECS 1
   IGNOREFIELDS YES
   ( "ID" POSITION( 00003:00014) CHAR(00012)
   , "DSN_NULL_IND_00002" POSITION( 00015) CHAR(1)
   , "DATA"
   POSITION( *) BLOB
   NULLIF(DSN_NULL_IND_00002)=X'FF'
   )
```

DSNU186I @ 364 14:29:26.91 DSNUGSRI - A EXPRESSION-BASED INDEX CANNOT
BE PROCESSED BY THIS UTILITY
DSNU012I 364 14:29:26.91 DSNUGBAC - UTILITY EXECUTION TERMINATED,
HIGHEST RETURN CODE=8

Similarly, CHECK DATA fails for the same reason. Listing 37 shows the CHECK DATA statement and its output.

Listing 37. CHECK DATA on JSON Data

```sql
CHECK DATA TABLESPACE DSN00003.JSONPO SCOPE ALL
   SHRLEVEL REFERENCE
```

DSNU186I @ 364 14:34:16.60 DSNUGSRI - A EXPRESSION-BASED INDEX CANNOT
BE PROCESSED BY THIS UTILITY
DSNU012I 364 14:34:16.60 DSNUGBAC - UTILITY EXECUTION TERMINATED,
HIGHEST RETURN CODE=8
Other support

In addition to the basic features already mentioned, DB2 for z/OS also provides the following XML support.

- XML schema validation
- XML transformation (XMLTABLE(), XSLT transformation)
- XML construction (XML publishing functions, XQuery constructor)

Currently, JSON does not have the variety of features that XML has in DB2 for z/OS, but this may change in the future.

Summary

This article provided information to help you decide when to select XML or JSON. Examples show similarities and differences when creating, querying, updating, and managing JSON and XML in DB2 for z/OS.

Acknowledgments

Thanks to Steve Chen, Dawn Zhao, Tim Hahn, and Romney White for their comments and assistance with this article.
Related topics

- **Leveraging DB2 9 for z/OS pureXML Technology** (Zhang, 2008): This white paper discusses the business value DB2 9 pureXML brings and some details of the XML features.
- Refer to **DB2 Version 10 for z/OS JSON Enabling PTF PM98357**.
- Read the:
  - DB2 Version 9.1 for z/OS XML Guide (SC18-9858)
  - DB2 Version 10 for z/OS pureXML Guide (SC19-2981-02)
  - DB2 Version 10 for z/OS SQL Reference (SC19-2983-02)
- Get an **Introduction to JSON**.
- Refer to **DB2 Version 10 for z/OS and JSON capability functions INFO APAR II14727**.
- **DB2 JSON Capabilities, Part 1: Introduction to DB2 JSON** (developerWorks, 2013): Introduces the DB2 JSON technology.
- **DB2 JSON capabilities, Part 2: Using the command-line processor** (developerWorks, 2013): How to set up and use the DB2 JSON command-line interface.
- **DB2 JSON capabilities, Part 3: Writing applications with the Java API** (developerWorks, 2013): Manage JSON documents - with and without transactions.
- **DB2 JSON capabilities, Part 4: Using the IBM NoSQL Wire Listener for DB2** (developerWorks, 2013): Get an introduction to the IBM NoSQL Wire Listener for DB2.

© Copyright IBM Corporation 2014
**Trademarks**
(www.ibm.com/developerworks/ibm/trademarks/)